Modular Language Product Lines

A Graph Transformation Approach

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain

Paolo Bottoni
Sapienza University of Rome
Rome, Italy

ABSTRACT

Modelling languages are intensively used in paradigms like model-driven engineering to automate all tasks of the development process. These languages may have variants, in which case the need arises to deal with language families rather than with individual languages. However, specifying the syntax and semantics of each language variant separately is costly, hinders reuse across variants, and may yield inconsistent semantics between variants.

To attack this problem, we propose a novel, modular way to describe product lines of modelling languages. Our approach is compositional, enabling the incremental definition of language families by means of modules comprising meta-model fragments, graph transformation rules, and rule extensions. Language variants are configured by selecting the desired modules, which entails the composition of a language meta-model and a set of rules defining its semantics. This paper describes a theory able to check consistent semantics among all languages within the family, an implementation as an Eclipse plugin, and an evaluation reporting drastic specification size reduction w.r.t. an enumerative approach.

CCS CONCEPTS

• **Software and its engineering** → **Semantics**; **Syntax**; **Model-driven software engineering**.

KEYWORDS

Model-driven engineering, Graph transformation, Product lines, Software language engineering.

ACM Reference Format:
Juan de Lara, Esther Guerra, and Paolo Bottoni. 2022. Modular Language Product Lines: A Graph Transformation Approach. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552444

1 INTRODUCTION

Modelling languages are ubiquitous in many engineering disciplines to describe manageable abstractions (models) of real, complex phenomena. This is no exception for software engineering, where paradigms like model-driven engineering (MDE) [3] make intensive use of modelling languages and models to conduct and provide automation for all phases of the development process. These models are specified via modelling languages, often domain-specific [21].

Modelling languages comprise abstract syntax (the concepts covered by the language), concrete syntax (their representation), and semantics (their meaning). In MDE, the abstract syntax of modelling languages is described by a meta-model; the concrete syntax by a model describing the rendering of the language elements; and the semantics by model transformations.

Sometimes, languages that share commonalities are organised into families. This is the case, for example, of the more than 120 variations of architectural languages reported in [30], and the many variants of Petri nets [34], access control languages [20] and symbolic automata [7] proposed in the literature. Likewise, language families can be defined to account for variants of a language directed to different kinds of user, or contexts of use. For example, within UML, simple versions of class diagrams are suitable for novices; complete versions for experts; and restricted ones (e.g., with single inheritance) for detailed design targeting languages like Java. However, describing the syntax and semantics of each language variant separately is costly, does not benefit from reuse across variants, and may yield inconsistent semantics between variants.

To tackle this issue, product lines [40] have been applied to the engineering of modelling languages [33]. Product lines permit the compact definition of a potentially large set of products that share common features. Hence, earlier works have created product lines of meta-models [8, 16] and model transformations [9, 44]. However, the former product lines [8, 16] do not consider semantics, while the latter do not support meta-model variants [44], are hard to extend [9], or are not based on formalisms that enable asserting consistency properties over the language family [9, 32].

In this work, we propose a novel modular approach for defining language product lines, which considers semantics and ensures semantic consistency across all members of the language family. The approach is based on modules that encapsulate a meta-model fragment and graph transformation rules [14]. Modules can also declare different kinds of dependencies on other modules, and extend the rules defined in those other modules. Overall, our approach enables the definition of a large set of language variants in a compact way, and the theory ensures the semantic consistency of each variant. To demonstrate the practicality of our proposal, we report on its realisation on a concrete tool called Capone [5] and on an evaluation that shows its benefits over an enumerative approach.

Paper organization. Sec. 2 motivates our proposal via a running example. Sec. 3 overviews our approach. Sec. 4 introduces the structure of our product lines. Sec. 5 expands them to consider behaviour using rules, and includes the main result of our theory about behaviour consistency. Sec. 6 reports on supporting tool, and Sec. 7

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552444

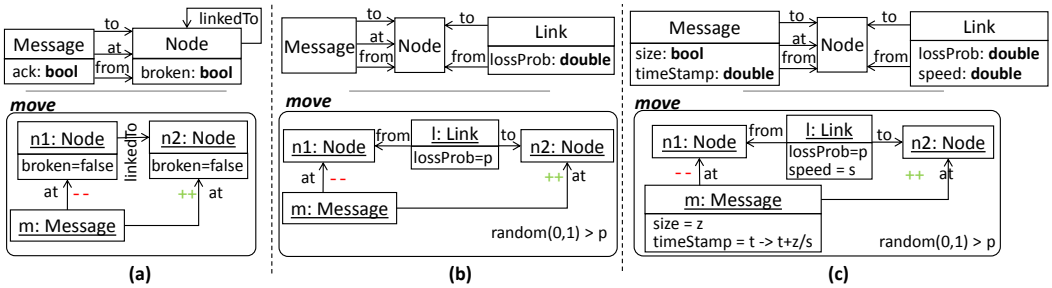

Figure 1: Three network language variants, and one example rule of each. (a) Simple links with node failures and acks. (b) Rich links with communication failures. (c) Rich links with communication failures and time. Rules use a compact notation with ++ denoting element creation, -- element deletion, and -> attribute change.

on an evaluation. Sec. 8 compares with related work, and Sec. 9 concludes the paper. An appendix includes the proof of Theorem 5.10.

2 MOTIVATION AND RUNNING EXAMPLE

We motivate our approach based on a family of domain-specific languages (DSLs) to model communication networks, composed of nodes that exchange messages with each other. We would like to support different usages of the language family, such as: study the behaviour of networks with node failures, deal with message loss probabilities, consider protocols and time performance, among others. As usual in MDE, we represent the language syntax with meta-models, and the semantics via (graph) transformation rules [14].

Fig. 1 shows the meta-model of three language variants within the family, and one example rule capturing the behaviour of each one. Variant (a) is for a language with simple links between nodes (reference linkedTo), supporting node failures (broken) and a simple protocol (ack). Variant (b) features rich node links (class Link) with a probability simulating communication loss (lossProb). Variant (c), in addition, includes timestamps and considers the speed of links.

A naive approach would define separate meta-models and transformation rules for each language variant. However, as Fig. 1 shows, the meta-models and rules share commonalities which one may not want to replicate. Moreover, the language family may need to evolve and be extended over time, so adding new language features to the family should be easy. However, in a naive approach, incorporating an optional feature (e.g., support for ad-hoc networks) implies duplicating all language variants and adding the new feature to the duplicates. This entails a combinatorial explosion of variants. Finally, evolving the rules of each variant separately may easily lead to inconsistencies between them.

An alternative solution to tackle our example would be to create one language that incorporates all possible features. However, this solution is not suitable either, as the language users would need to deal with an unnecessarily complex language, when a simpler variant would suffice. Moreover, some language features may be incompatible if they represent alternative options (e.g., a network should not have both simple and rich links at the same time).

A sensible solution to define and manage a family of languages, like the one described, should meet the following requirements:

(1) Succinctness: Specifying a DSL family should require much less effort than specifying each language variant in isolation.

(2) Extensibility: Adding a new language variant to the family should be easy, and should not require changing other existing variants. This allows incremental language construction.

(3) Reusability: The specifications of language variants should be as reusable as possible, to minimise effort and avoid duplications.

(4) Analysability: It should be possible to analyse a language family to ensure that the behaviour of its variants is compatible with the base behaviour of the language.

In the following, we propose a novel approach to define modelling language variants that satisfies these requirements. It enables a compact, extensible specification of the syntax of a language family (Sec. 4), and a compact, extensible specification of semantics that ensures consistency across all members of the family (Sec. 5). First, the next section provides an overview of the approach.

3 OVERVIEW OF THE APPROACH

Fig. 2 shows a scheme of our solution to define families of languages, whereby each language feature is defined as a module comprising a meta-model fragment for the syntax, and a set of graph transformation rules for the semantics.

A set of modules $M_1,...,M_n$ may extend another module M. In such a case, module M is said to be a *dependency* of $M_1,...,M_n$, and modules $M_1,...,M_n$ are its *extensions*. The extensions $M_1,...,M_n$ need to specify their role in the dependency (label 1 in Fig. 2). The possible roles are the standard variability options in feature modelling [19]: optional (the extension can be present or not in a language variant that includes the dependency), alternative (exactly one of the possible alternative extensions must be present), OR (one or more of the

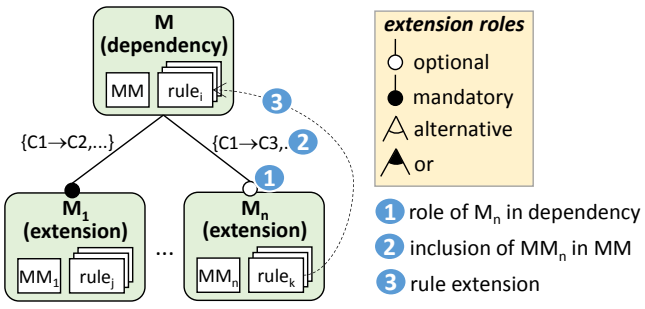

Figure 2: Scheme of a modular language product line.

OR extensions can be present), or mandatory (if the dependency is present, so must be the extension). The extension also needs to specify how to merge its structure (its meta-model) with the one in the dependency module (label 2), and which of its rules extend rules in the dependency, if there is any (label 3).

Then, we define a *modular language product line* (LPL) as a tree of modules, with relations from the extensions to their dependencies, and one identified root module. A language variant can be obtained from the LPL by making a selection of modules that satisfy the dependencies. This induces proper compositions of the meta-models and rules in the selected modules.

This approach satisfies the requirements of the previous section:

(1) Succinctness: The use of product lines [35, 40] avoids defining each language variant in isolation. Instead, modules describe simpler language features, which can be combined to obtain the desired language variant.

(2) Extensibility: Taking inspiration from practical component-based systems like OSGI [36] or Eclipse [13], our modules encapsulate syntax and semantics, and can extend another module. This results in an extensible design of languages, since adding a new module to the LPL does not imply modifying a global structure – like a "monolithic" 150% meta-model overlapping the meta-model of all language variants [16], or a "global" feature model describing all language variants [19].

(3) Reusability: Extension modules can reuse the syntax and semantics declared in their dependencies.

(4) Analysability: We rely on graph transformation to express the semantics of modules. As Sec. 5 will show, this enables consistency checking across all members of a language family.

4 LANGUAGE PRODUCT LINES: STRUCTURE

Next, we present the formalization of our approach. We start by considering the abstract syntax of the languages, while Sec. 5 will expand the notion of module with rules to express behaviour. We use graphs to encode models and meta-models, using the notion of E-graph. An E-graph [14] is defined by two sets of nodes – graph and data nodes – and three kinds of edges: the regular graph edges connecting two graph nodes, and special edges for node and edge attribution (i.e., connecting graph nodes and edges, to data nodes).

Definition 4.1 (Language Module). A language module is defined by a tuple $M = \langle MM, M_D, RO, IN, \Psi \rangle$, where:

- MM is a meta-model;
- M_D is a module, called dependency;
- $RO \in \{ALT, OR, OPT, MAN\}$ is the role of M in the dependency, one among *alternative*, *OR*, *optional*, and *mandatory*.
- $IN = MM \longleftarrow C \longrightarrow MM(M_D)$ is an inclusion span between MM and the meta-model of M's dependency.
- Ψ is a boolean formula that uses modules as variables.

M is called *top* if $M_D = M$ and $\Psi = true$. We use predicate $top(M)$ to identify top modules: $top(M) \iff M_D(M) = M \wedge \Psi(M) = true$.

In this and subsequent definitions, we use the following notation. Given a module M_i, we use $MM(M_i)$ for the meta-model of M_i, and similarly for the other components of M_i (i.e., M_D, RO, IN, Ψ). $DEP^+(M_i)$ denotes the transitive closure of its dependencies (i.e., its dependency, the dependency of its dependency, etc.). $DEP(M_i) = DEP^+(M_i) \setminus \{M_i\}$ is the transitive closure excluding itself, which is empty in top modules, and equals to DEP^+ in non-top modules. $DEP^*(M_i) = DEP^+(M_i) \cup \{M_i\}$ is the reflexive transitive closure (i.e., including the module M_i as well). Note that, typically, IN is the identity inclusion for top modules.

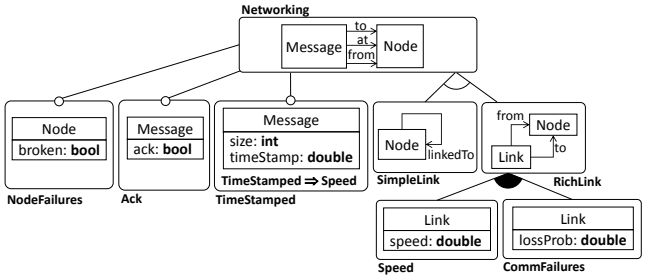

Figure 3: Language product line for the example.

An LPL is a set of modules with a single top (1), closed under the modules' dependencies (2), and without dependency cycles (3).

Definition 4.2 (Language Product Line). A language product line $LPL = \{M_i\}_{i \in I}$ is a set of modules s.t.

$$\exists_1 M_i \in LPL \cdot top(M_i) \wedge \quad (1)$$

$$\forall M_i \in LPL \cdot M_D(M_i) \in LPL \wedge \quad (2)$$

$$M_i \in DEP^+(M_i) \implies top(M_i) \quad (3)$$

Example. Fig. 3 shows the LPL for the running example. It comprises 8 modules: Networking, NodeFailures, Ack, TimeStamped, SimpleLink, RichLink, Speed, and CommFailures. Each module shows its meta-model inside. For extensibility, dependencies are expressed from a module (e.g., Ack) to its dependency (e.g., Networking). This permits adding new modules to the LPL without modifying existing ones. The figure omits the dependency of the top module (Networking) and span IN is implicit, given by the equality of names in meta-model elements in both an extension module and its dependency. For example, Message in Ack is mapped to Message in Networking. Module TimeStamped has a formula Ψ stating that, if a language variant includes the module TimeStamped, it must also include the module Speed. For clarity, the figure omits the formula Ψ when it is $true$, as is the case for all modules but TimeStamped.

We use $TOP(LPL)$ to denote the only top module in LPL. Given a module $M_i \in LPL$, we define the sets $X(M_i) = \{M_j \in LPL \mid M_D(M_j) = M_i \wedge RO(M_j) = X)\}$ for $X \in \{ALT, OR, OPT, MAN\}$, to obtain the extension modules of M_i with role X.

Example. The top module of the LPL of Fig.3 is Networking. For this module, we have ALT(Networking) = {SimpleLink, RichLink}, OPT(Networking) = {NodeFailures, Ack, TimeStamped}, while MAN(Networking) and OR(Networking) are empty.

Given an LPL, a specific language of the family can be obtained by choosing a valid configuration of modules, as per Def. 4.3.

Definition 4.3 (Language Configuration). Given a language product line LPL, a configuration $\rho \subseteq LPL$ is a set of modules s.t.:

$$TOP(LPL) \in \rho \wedge \quad (1)$$

$$M \in \rho \implies (\forall M_i \in MAN(M) \cdot M_i \in \rho \wedge \quad (2)$$

$$ALT(M) \neq \emptyset \implies \exists_1 M_i \in ALT(M) \cdot M_i \in \rho \wedge \quad (3)$$

$$OR(M) \neq \emptyset \implies \exists M_i \in OR(M) \cdot M_i \in \rho \wedge \quad (4)$$

$$M_D(M) \in \rho) \wedge \quad (5)$$

$$\bigwedge_{M_i \in LPL} \Psi(M_i)[true/\rho, false/(LPL \setminus \rho)] = true \quad (6)$$

We use $CFG(LPL)$ to denote the set of all configurations of LPL.

A configuration should contain the top module of the LPL (1), and if a configuration includes a module, then it should also include: all its mandatory extension modules (2), exactly one of its alternative extension modules (3), at least one of its OR extension modules (4), and its dependency (5). Recall that the top module has itself as its only dependency. Finally, the formula of all modules in the LPL should evaluate to true when substituting the modules that the configuration includes by true, and the rest by false (6).

Example. The LPL of Fig. 3 admits 24 configurations, including ρ_0 = {Networking, SimpleLink} (the smallest configuration), ρ_1 = {Networking, SimpleLink, NodeFailures, Ack}, ρ_2 = {Networking, RichLink, CommFailures}, and ρ_3 = {Networking, RichLink, CommFailures, TimeStamped, Speed}. Due to Ψ(TimeStamped), a configuration that selects TimeStamped must select Speed as well. Configurations can include *zero or more* modules of OPT(Networking), and must include *one or more* modules of OR(RichLink) when RichLink is selected.

Given a configuration ρ, we derive a *product meta-model* by merging the meta-models of all modules in ρ, using the inclusion spans as glueing points. This is formalized using the categorical notion of co-limit [26].

Definition 4.4 (Derivation). Given a language product line LPL and a configuration $\rho \in CFG(LPL)$, a product meta-model MM_ρ is given by the co-limit object of all meta-models and spans in the set $\{IN(M_i) = \langle MM(M_i) \longleftarrow C \longrightarrow MM(M_D(M_i))\rangle \mid M_i \in \rho\}$.

Remark. Since we use a simple notion of meta-model (an E-graph), we do not require well-formedness conditions of the derived meta-model. Richer notions of meta-model, e.g., with inheritance or named attributes, would require such conditions to avoid, e.g., inheritance cycles or repeated attribute/reference names.

Example. Figs. 1(a), 1(b) and 1(c) from Sec. 2 show the product meta-models MM_{ρ_1}, MM_{ρ_2} and MM_{ρ_3}, respectively.

Given a language product line LPL and a module $M \in LPL$, we need to derive the meta-model used to type the rules of M. This meta-model – called the *effective meta-model* of M – is composed out of the meta-models of the modules included in all configurations that include M. Hence, we define the set $CDEP(M) = \bigcap_{\rho_i \in CFG(LPL) \cdot M \in \rho_i}$, which is the intersection of all configurations that include M, and comprises the explicit module dependencies of M (i.e., $DEP^*(M)$) and other implicit dependencies due to the formula Ψ in modules. Then, the effective meta-model of M, written $EFF(M)$, is $MM_{CDEP(M)}$, calculated as in Def. 4.4 but using $CDEP(M)$ instead of a configuration ρ. Intuitively, it is the common slice of any product meta-model MM_ρ s.t. $M \in \rho$.

Example. The effective meta-model of CommFailures is MM_{ρ_2} in Fig. 1(b), as $CDEP$(CommFailures) = {CommFailures, RichLink, Networking}. In turn, $CDEP$(TimeStamped) = {TimeStamped, Networking, Speed, RichLink}, since Speed is in every configuration that includes TimeStamped – due to the formula in the latter module – while RichLink belongs to any configuration containing Speed.

We purposely mix the product space (i.e., the modules) and the variability space (i.e., the feature model). One can see our modules as features, and derive a feature model from their dependencies, which then can be used to select a configuration, as Sec. 6 shows.

5 LANGUAGE PRODUCT LINES: BEHAVIOUR

Next, we extend LPLs with behaviour. First, Sec. 5.1 defines *rules* and *extension rules*, which Sec. 5.2 uses to extend modules and LPLs with behaviour. Then, Sec. 5.3 analyses the conditions for a behavioural LPL to define a consistent behaviour, where the behaviour of every language does not contradict that of simpler language versions.

5.1 Rules and Extension Rules

We use graph transformation rules – following the double pushout approach [14] – to express module behaviour. In this approach, a rule is defined by a span of three graphs: a left-hand side graph L, a right-hand side graph R, and a kernel graph K identifying the elements of L and R that the rule preserves. In addition, a rule has a set of negative application conditions (NACs), as Def. 5.1 shows.

Definition 5.1 (Graph Transformation Rule). A rule $r = \langle L \xleftarrow{l} K \xrightarrow{r} R, NAC = \{L \xrightarrow{n_i} N_i\}_{i \in I}\rangle$ is made of an injective span of (E-graph) morphisms, and a set of negative application conditions as injective (E-graph) morphisms.

Remark. We assume rules over typed E-graphs, where L, K and R have a type morphism to a common meta-model.

Example. Fig. 4(a) shows a rule example that follows Def. 5.1. Morphisms l and r identify elements with equal name. The rule is applicable when a non-broken node has a message and is connected to another non-broken node (graph L). Applying the rule deletes the edge from the message to the first node (graph K) and creates an edge from the message to the second node (graph R). We will use a more compact notation, used in tools like Henshin [1] (cf. Fig. 4(b)) where all graphs L, K, R and N_i are overlapped. Elements in $L \setminus K$ (those deleted) are marked with −−, those in $R \setminus K$ (those created) are marked with ++, and those in a NAC (those forbidden) are marked with !! plus a subindex if there are several NACs.

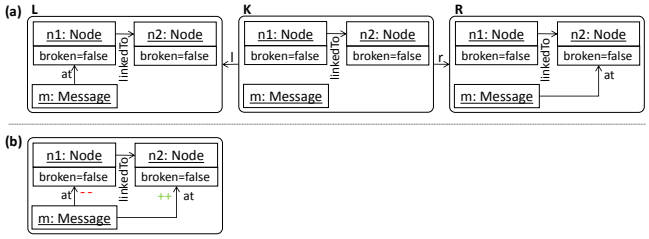

Figure 4: Rule using Def. 5.1 (a) and compact notation (b).

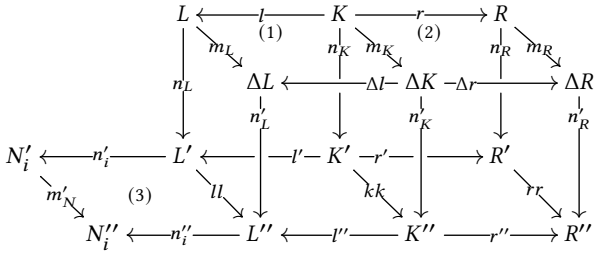

Figure 5: Δ-rule application to a rule.

To enable the reuse of the rule-based behaviour defined for one module in its extensions, we propose a mechanism for rule extension based on higher-order rules [46]. Our *extension rules* add elements to a base rule. We support two kinds of extension rules: adding elements to L, K and R (called Δ-rules), and adding extra NACs (called NAC-rules). Def. 5.2 is for Δ-rules.

Definition 5.2 (Δ-rule). A Δ-rule $\Delta_r = \langle L \xleftarrow{l} K \xrightarrow{r} R, \Delta L \xleftarrow{\Delta l} \Delta K \xrightarrow{\Delta r} \Delta R, m = \langle m_X : X \to \Delta X \rangle \text{ (for } X \in \{L, K, R\})\rangle$ is composed of two spans and a triple $m = \langle m_L, m_K, m_R \rangle$ of injective morphisms s.t. all squares commute ($m_L \circ l = \Delta l \circ m_K$, $m_R \circ r = \Delta r \circ m_K$).

A Δ-rule is applied to a standard rule via three injective morphisms, performing pushouts (POs), as per Def. 5.3. A PO is a glueing construction that merges two graphs by the common elements identified by another one [26].

Definition 5.3 (Δ-rule Application). Given a Δ-rule Δ_r, a rule r', and a triple $n = \langle n_X : X \to X' \rangle$ (for $X \in \{L, K, R\}$) of injective morphisms s.t. the back squares (1) and (2) in Fig. 5 commute, then Δ_r is applied to r' (written $r' \overset{\Delta_r}{\Longrightarrow} r''$) yielding rule $r'' = \langle L'' \xleftarrow{l''} K'' \xrightarrow{r''} R'', NAC'' = \{L'' \xrightarrow{n_i''} N_i''\}_{i \in I} \rangle$, which is built as follows:

(1) Span $L'' \xleftarrow{l''} K'' \xrightarrow{r''} R''$ is obtained by the POs of the spans $X' \xleftarrow{n_X} X \xrightarrow{m_X} \Delta X$ (for $X \in \{L, K, R\}$), where morphisms l'' and r'' exist due to the universal PO property (over K''[1]).

(2) The set NAC'' is obtained by the POs of the spans $N_i' \xleftarrow{n_i'} L' \xrightarrow{ll} L''$ (square (3) in Fig. 5) for every $L' \xrightarrow{n_i'} N_i'$ in NAC'.

A Δ-rule Δ_r' can also be applied to another Δ-rule Δ_r yielding a composed Δ-rule that performs the actions of both Δ-rules. The main idea is to match Δ_r' on $\Delta L \leftarrow \Delta K \rightarrow \Delta R$, and compose the actions of both rules, as Def. 5.4 shows.

Definition 5.4 (Δ-rule Composition). Let Δ_r and Δ_r' be two Δ-rules, and $n = \langle n_X : X' \to \Delta X \rangle$ (for $X = \{L, K, R\}$) be a triple of morphisms s.t. the squares (1) and (2) (at the top) in Fig. 6 commute. The composition of Δ_r' and Δ_r through n, written $\Delta_r' +_n \Delta_r$, is a Δ-rule $\langle L \xleftarrow{l} K \xrightarrow{r} R, \Delta L'' \xleftarrow{\Delta l''} \Delta K'' \xrightarrow{\Delta r''} \Delta R'', m'' = \langle m_L'', m_K'', m_R'' \rangle\rangle$ where the span $\Delta L'' \xleftarrow{\Delta l''} \Delta K'' \xrightarrow{\Delta r''} \Delta R''$ results from the POs of spans $\Delta X \xleftarrow{n_X} X' \xrightarrow{m_X'} \Delta X'$ (for $X \in \{L, K, R\}$) [2], and $m_X'' : X \to \Delta X = \Delta m_X \circ m_X$ (for $X = \{L, K, R\}$).

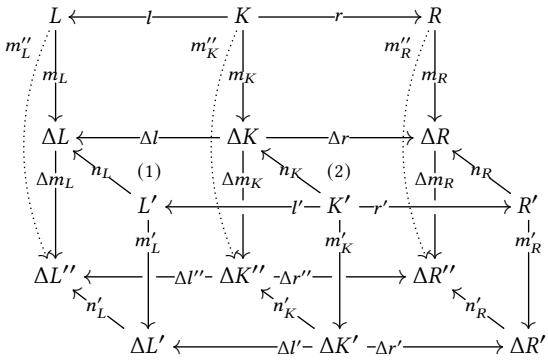

Figure 6: Δ-rule composition.

Given a Δ-rule Δ_r and a Δ- or standard rule r, we write $n : \Delta_r \to r$ to denote the morphism triple between Δ_r and r.

NAC-rules rewrite standard rules by adding them a NAC.

Definition 5.5 (NAC-rule and Application). A NAC-rule $N_r = \langle n : NL \to N \rangle$ consists of an injective morphism.

Given a rule r and an injective morphism $m : NL \to L$, applying N_r to r via m (written $r \overset{N_r}{\Longrightarrow} r'$) yields $r' = \langle L \xleftarrow{l} K \xrightarrow{r} R, NAC \cup \{L \xrightarrow{n'} N'\}\rangle$, where N' is the PO object of $N \xleftarrow{n} NL \xrightarrow{m} L$.

Example. Fig. 7 shows examples of Δ- and NAC-rule applications. Fig. 7(a) shows a standard rule move defined on the context of module Networking. Fig. 7(b) shows a Δ-rule, called Δmove-rl, used to increase a rule having two nodes with a link between them. We use a compact notation for Δ- and NAC-rules, where the added elements are enclosed in regions labelled as Δ followed by the place of addition: {preserve} for L, K and R; {delete} for L; {create} for R; and {forbid} for adding a new NAC. This way, the Δ-rule in Fig. 7(b) adds a Link node and two edges to the L, K and R of a rule. Similarly, the Δ-rule in Fig. 7(c) adds an attribute and a condition; and the NAC-rule in Fig. 7(d) adds a NAC.

For illustration of Def. 5.4, Fig. 7(e) shows the Δ-rule resulting from the composition of the Δ-rules (b) and (c), via the Link l. Please note that rule (c) is applied on the Δ{preserve} part of rule (b). The resulting Δ-rule performs all actions of the two Δ-rules. Fig. 7(f) is the rule resulting of applying Δ-rule (e) to the rule in Fig. 7(a), so that additional preserved elements are added to (a) (cf. Def. 5.3). The rule in Fig. 7(g) is the result of applying the NAC-rule (d) to the rule (f) twice (cf. Def. 5.5). This adds two NACs to (f) via two different morphisms: one identifying n with n1, and another identifying n with n2. These NACs are marked with $!!_1$ and $!!_2$.

Remark. The application of NAC- and Δ-rules to a given rule, and the composition of Δ-rules, are independent:

[1] l'' uniquely exists since we have $K' \xrightarrow{ll \circ l'} L'' \xleftarrow{n_L' \circ \Delta l} \Delta K$, and similarly for r''.

[2] Morphisms $\Delta l''$ and $\Delta r''$ uniquely exist by the universal PO property of $\Delta K'' : \Delta l''$ uniquely exists since $\Delta K \xrightarrow{\Delta m_L \circ \Delta l} \Delta L'' \xleftarrow{n_L' \circ \Delta l'} \Delta K'$, and similarly for $\Delta r''$.

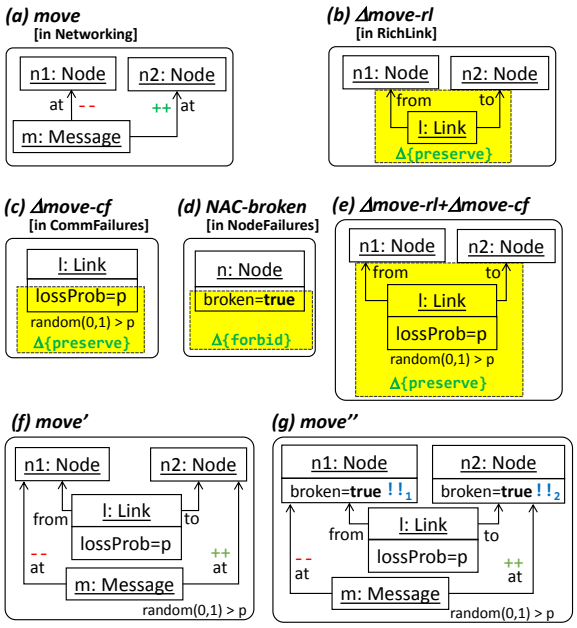

Figure 7: (a) Rule from module Networking. (b) Δ-rule from RichLink. (c) Δ-rule from CommFailures. (d) NAC-rule from module NodeFailures. (e) Δ-rule resulting from composing Δ-rules (b) and (c). (f) Result of applying Δ-rule (e) to rule (a). (g) Result of applying NAC-broken to rule (f) twice.

- Given a rule r and a set $D = \{m_i \colon \Delta_{r_i} \to r\}_{i \in I}$ of morphism triples from Δ-rules into r, we can apply each Δ-rule in D to r in any order, yielding the same result since Δ-rules are non-deleting and there is no forbidding context for their application. We use the notation $r \overset{D}{\Longrightarrow} r'$ for the sequential application $r \overset{\Delta_{r_0}}{\Longrightarrow} r_0 \overset{\Delta_{r_1}}{\Longrightarrow} \dots r'$ of each Δ-rule in D starting from r.
- The composition of a set of Δ-rules (Δ_{r_i}) with a Δ-rule (Δ_r) is independent of the application order. Given a set $D = \{m_i \colon \Delta_{r_i} \to \Delta_r\}_{i \in I}$, $\amalg_D$ denotes the Δ-rule that results from composing each Δ_{r_i} (through m_i) to Δ_r in sequence.
- The application of NAC-rules is independent of the application order. Given a set N of morphisms from NAC-rules into a rule r, we write $r \overset{N}{\Longrightarrow} r'$ to denote the sequential application of each NAC-rule in N starting from r.

5.2 Behavioural Language Product Lines

To add behaviour to LPLs, we incorporate into modules a set R of rules, two sets ΔR and NR of extension rules (called Δ-rules and NAC-rules), and two sets EX and NEX of morphisms from the extension rules to (standard and Δ-) rules in the module dependencies.

Definition 5.6 (Behavioural Module). A behavioural module extends Def. 4.1 of language module as follows. A behavioural module $M = \langle MM, M_D, RO, IN, \Psi, R, \Delta R, NR, EX, NEX \rangle$ consists of:

- MM, M_D, RO, IN and Ψ as in Def. 4.1
- Sets $R = \{r_i\}_{i \in I}$ of rules, $\Delta R = \{\Delta_{r_j}\}_{j \in J}$ of Δ-rules, and $NR = \{N_{r_k}\}_{k \in K}$ of NAC-rules, all typed by the effective meta-model of M, $EFF(M)$
- A set $EX = \{m_{ij} \colon \Delta_{r_i} \to r_j \mid \Delta_{r_i} \in \Delta R \wedge r_j \in (R(M_j) \cup \Delta R(M_j)) \wedge M_j \in DEP(M)\}$ of morphism triples m_{ij} mapping each $\Delta_{r_i} \in \Delta R$ to at least a (Δ- or standard) rule r_j in some module M_j of M's dependencies
- A set $NEX = \{m_{ij} \colon NL_i \to L_j \mid N_{r_i} \in NR \wedge r_j \in R(M_j) \wedge M_j \in DEP(M)\}$ of morphisms m_{ij} mapping each NAC-rule $N_{r_i} = \langle n_i \colon NL_i \to N_i \rangle \in NR$ to at least a rule r_j (with LHS L_j) in some module M_j of M's dependencies

Definition 5.6 uses $R(M_j)$ (resp. $\Delta R(M_j)$) to refer to set R (resp. ΔR) within the behavioural module M_j. In the remainder of the section, we will use a similar notation for the other components of behavioural modules. Since sets EX and NEX in Def. 5.6 contain morphisms to (Δ-)rules in $DEP(M)$, this entails that top modules cannot define extensions for their own rules.

We omit the definitions of behavioural LPL, configuration of a behavioural LPL and CFG since they are the same as in Defs. 4.2 and 4.3, only considering behavioural modules instead of modules. However, we need to provide a new notion of derivation that complements that of Def. 4.4 (yielding a meta-model) with rule composition via the extension rules (yielding a set of rules). First, we define the sets of extension rules that apply to a given rule.

Definition 5.7 (Rule Extensions). Given a behavioural language product line BPL, a behavioural module $M_i \in BPL$, a rule $r \in R(M_i)$, and a Δ-rule $\Delta_r \in \Delta R(M_i)$, we define the sets:

- $EX(\Delta_r) = \{m_j \colon \Delta_{r_j} \to \Delta_r \mid M_j \in BPL \wedge M_i \in DEP(M_j) \wedge m_j \in EX(M_j)\}$ of all morphism triples from every Δ-rule Δ_{r_j} rewriting Δ_r
- $CEX(r) = \{m_j \colon \amalg_{EX(\Delta_{r_j})} \to r \mid M_j \in BPL \wedge M_i \in DEP(M_j) \wedge \Delta_{r_j} \to r \in EX(M_j)\}$ of all morphism triples from every Δ-rule Δ_{r_j} (composed with all possible extensions in $EX(\Delta_{r_j})$) rewriting r
- $NEX(r) = \{m \colon NL \to L \mid M_j \in BPL \wedge M_i \in DEP(M_j) \wedge m \in NEX(M_j)\}$ of all morphisms from every NAC-rule N_r adding a NAC to r

Given a behavioural LPL and a configuration, we can perform a behavioural derivation. This yields the set of rules in the selected modules, extended by the rule extensions defined in those modules.

Definition 5.8 (Behavioural Derivation). Given a behavioural product line BPL and a configuration $\rho \in CFG(BPL)$, we obtain a set $R = \{r_i''\}_{i \in I}$ of rules, where each r_i'' is obtained by the rewriting $r_i \overset{CEX(r_i)}{\Longrightarrow} r_i' \overset{NEX(r_i)}{\Longrightarrow} r_i''$ of each rule $r_i \in \bigcup_{M_j \in \rho} R(M_j)$ defined by the modules included in the configuration. We sometimes use the notation $\rho(r_i)$ to refer to the resulting rule r_i'' above.

Example. Assume configuration ρ = {Networking, RichLink, CommFailures, NodeFailures} and the rules of Fig. 7. Then, $EX(\Delta\text{move-rl}) = \{\Delta\text{move-cf} \to \Delta\text{move-rl}\}$, and Fig. 7(e) shows the Δ-rule $\amalg_{EX(\Delta move-rl)}$. This way, $CEX(\text{move}) = \{\amalg_{EX(\Delta move-rl)} \to \text{move}\}$, and rule move' in Fig. 7(f) derives from move $\overset{CEX(move)}{\Longrightarrow}$ move'. Note that EX(RichLink) contains a morphism triple from Δmove-rl to move. Composing

Δmove-rl with all extensions in $EX(\Delta\text{move-rl})$ preserves such morphism triples (since composition adds elements to the Δ part of the rule only), which are then used in set $CEX(\text{move})$. Finally, $NEX(\text{move})$ contains two morphisms from NAC-broken in module NodeFailures to move. Thus, move'' (cf. Fig. 7(g)) is obtained by move' $\xRightarrow{NEX(move)}$ move''. The morphisms in *NEX*, from NAC-broken to move, are also valid morphisms from NAC-broken into move', since the previous derivation via *CEX* only adds elements to move.

5.3 Consistent Extensions

We distinguish a particular class of extension rules, called *modular extensions*, which only incorporate into another rule elements of meta-model types added by the module. That is, modular extensions do not add elements of types existing in the meta-model of a dependency, since this would risk changing the semantics of the extended rules. Instead, modular extensions "decorate" existing rules with elements reflecting the semantics of the new elements added to the meta-model.

Definition 5.9 (Modular Extension). Given a behavioural product line *BPL* and a behavioural module $M \in BPL$:

(1) A Δ-rule $\Delta_r \in \Delta R$ is a modular extension if each element in $\Delta X \setminus X$ (for $X \in \{L, K, R\}$) is typed by $MM \setminus C$ (for $IN = \langle MM \leftarrow C \rightarrow MM(M_D)\rangle$).
(2) A NAC-rule $N_r \in NR$ is a modular extension if each element in $N \setminus NL$ is typed by $MM \setminus C$ (for $IN = \langle MM \leftarrow C \rightarrow MM(M_D)\rangle$).
(3) Given a rule $r_i \in R$ and a rewriting $r_i \xRightarrow{CEX(r_i)} r_i' \xRightarrow{NEX(r_i)} \rho(r_i)$, we say that $\rho(r_i)$ is a modular extension of r if all rule extensions in $CEX(r_i)$ and $NEX(r_i)$ are modular extensions.

Given a Δ- or NAC-rule r, we use predicate $mod-ext(r)$ to indicate that r is a modular extension.

Remark. The composition of two modular extensions is a modular extension, by transitivity of Def. 5.9 (1,2).

Example. The Δ-rule Δmove-rl in Fig. 7(b) is a modular extension, since it adds a node of type Link and edges of types from and to, belonging to the meta-model of RichLink but not to that of Networking. This Δ-rule would not be a modular extension if it added, e.g., a Message node, as this may change the semantics of the base rule on models typed by the meta-model of Networking. Instead, Δmove-rl adds extra elements that only affect models typed by $EFF(\text{RichLink})$. Similarly, the NAC-rule NAC-broken is a modular extension since it adds an attribute of type broken, which belongs to the meta-model in NodeFailures but not to the one in Networking. Finally, rule move'' (Fig. 7(g)) is a modular extension of rule move (Fig. 7(a)), since Δmove-rl, Δmove-cf and NAC-broken are modular extensions.

Modularly extended rules become of special interest in our setting, since they do not change the semantics of the base rule in models conformant to simpler language versions. Theorem 5.10 (whose proof is in the appendix) captures this property.

Theorem 5.10 (Consistent Extension Semantics). Let: *BPL* be a behavioural LPL; $\rho \in CFG(BPL)$ be a configuration; $r \in R(M_i)$ with $M_i \in \rho$ be a rule in some behavioural module M_i of the configuration ρ; and G_ρ be a model typed by MM_ρ. Then, for every direct derivation $G_\rho \overset{\rho(r)}{\Longrightarrow} H_\rho$, there is a corresponding direct derivation $G \overset{r}{\Longrightarrow} H$, if $\rho(r)$ is a modular extension of r (and where G, G_ρ, H and H_ρ are models related as Fig. 8 shows).

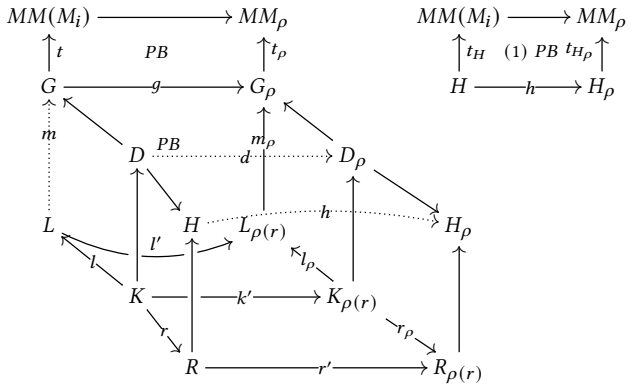

Figure 8: Consistent extension.

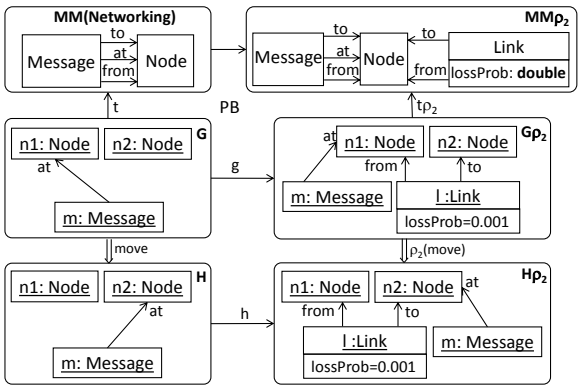

Figure 9: Consistent extension example: Applying rule ρ_2(move) to G_{ρ_2} implies that applying move to G is possible.

Example. Fig. 9 shows a consistent extension. Given the configuration $\rho_2 = \{\text{Networking, RichLink, CommFailures}\}$, the extended rule ρ_2(move) (Fig. 7(f)) is only applicable to a model (like G_{ρ_2} in the figure) if rule move (the base rule in Networking) is applicable to the model deprived of the elements introduced by MM_{ρ_2}.

The next corollary summarizes the implications of Theorem 5.10.

Corollary 5.11. Given a behavioural LPL *BPL*, a configuration $\rho \in CFG(BPL)$, and a rule $r \in R(M_i)$ with $M_i \in \rho$ and $mod-ext(\rho(r))$:

(1) $\rho(r)$ does not delete more elements with types of $EFF(M_i)$ than r (implied by item (2) in the proof of Theorem 5.10).
(2) $\rho(r)$ does not create more elements with types of $EFF(M_i)$ than r (implied by item (3) in the proof of Theorem 5.10).
(3) $\rho(r)$ is not applicable more often than r (implied by item (1) in the proof of Theorem 5.10).

Finally, we define consistent behavioural LPLs as those where all extension rules of each module are modular extensions, and only the top module defines rules.

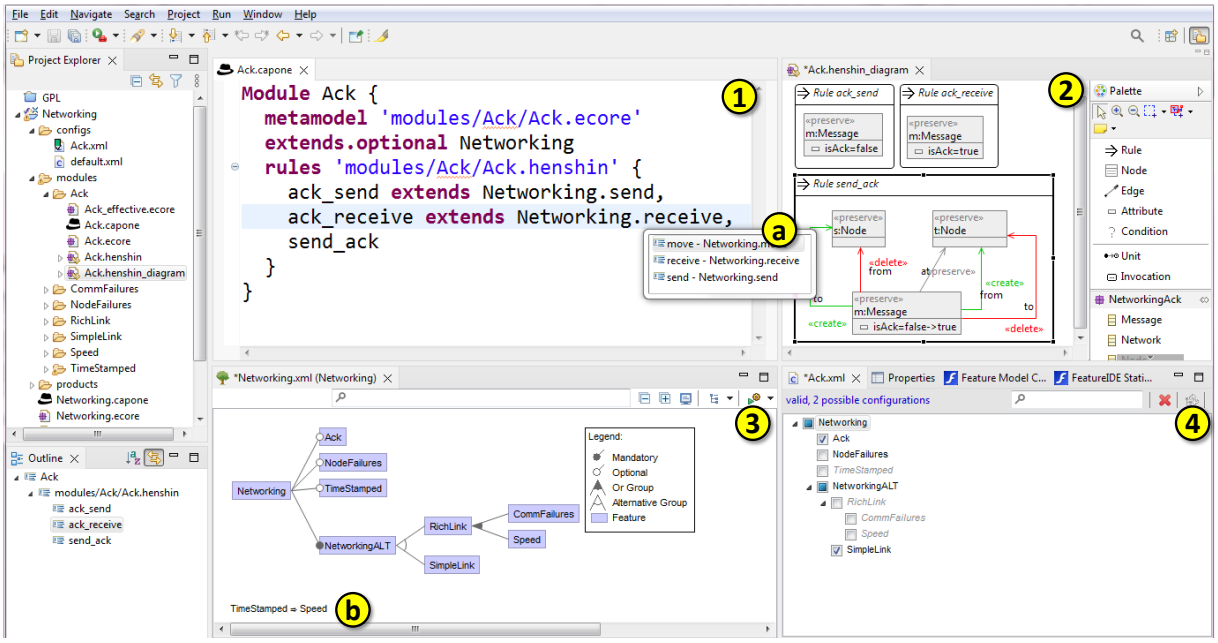

Figure 10: Capone in action: (1) Defining a module. (2) Module's rules. (3) Derived feature model. (4) Selecting a variant.

Definition 5.12 (Consistent Behavioural LPL). A behavioural LPL BPL is consistent if $\forall M_i \in BPL \cdot \neg top(M_i) \implies (R(M_i) = \emptyset \wedge \forall r_i \in \Delta R(M_i) \cup NR(M_i) \cdot mod - ext(r_i))$.

Consistent LPLs do not allow language variants to incorporate new actions (i.e., new rules) in the semantics of the top module, and all extensions are required to be modular. Even if this requirement might be too strong for some language families, the result permits controlling and understanding potential semantic inconsistencies between language variants. We leave for future work the investigation of finer granular notions of (in-)consistency.

Example. The running example is not a consistent LPL. The top module declares three rules: send, move and receive. While all modules define modular extensions, module Ack needs to introduce a new rule to send back ack messages (cf. Fig. 10). Using our approach, the designer of the language family can identify the non-consistent variants with the base behaviour and the reasons for inconsistency.

6 TOOL SUPPORT

All the concepts above are realised in Capone (Component-bAsed PrOduct liNEs), an Eclipse plugin relying on EMF [43] as the modelling technology and Henshin [1] for the rules, and extending FeatureIDE [31], a framework to construct product-line solutions, so as to support composition of language modules, Henshin rules, and EMF meta-models for a language configuration selection [5] .

Fig. 10 shows Capone in action. The view with label 1 shows the definition of a module using a domain-specific language designed by us, and realised using the Xtext framework [49]. The editor permits declaring the meta-model fragment referencing an existing *ecore* file, and instead of requiring explicit meta-model mappings, it relies on equality of names (of classes, attributes, references) for meta-model merging. Modules can also declare a formula and a dependency, and refer to a *henshin* file with their rules (see view with label 2 for an example *henshin* file). The editor suggests possible rules to extend, obtained from the module's dependencies recursively (shown in the pop-up window with label *a*). To compose Henshin rules, Capone relies on equality of identifiers. Interestingly, the implementation does not need to distinguish NAC- from Δ-rules, since NACs in Henshin are expressed as elements tagged as forbid.

The views with labels 3 and 4 are contributed by FeatureIDE. The view with label 3 contains the feature model capturing the module structure of the LPL. Our tool generates this model automatically out of the module structure. Feature models in FeatureIDE are more restricted than the ones we support. In particular, features cannot mix groups of OR/alternative children features with other types of features. Hence, the resulting feature model introduces intermediate features for this (cf. NetworkingALT in the view with label 3). Finally, modules' Ψ formulae are added as cross-tree constraints of the feature model (cf. label (b) in the figure).

Overall, users can select a specific configuration of the generated feature model (view 4). Then, Capone uses this configuration to merge the meta-models and rules corresponding to the language variant selected. Alternatively, it is possible to generate the meta-models and rules for all language variants.

7 EVALUATION

Next, we aim at answering two research questions (RQs), which assess the satisfaction of requirements 1 and 2 stated in Sec. 2:

RQ1 What is the effort reduction of the approach w.r.t. an explicit definition of each language variant?
RQ2 What is the typical effort for adding a new feature to an LPL?

To answer these RQs, we compared 4 language families built using our approach, w.r.t. an enumerative approach that would create each language variant separately from scratch.

Table 1: Metrics comparing LPLs (left) and behavioural LPLs (right) with respect to an enumerative approach.

		LPL (structure)		Enumerative app.		Behavioural LPL		Enumerative app.		
Name	**# Configs**	**# Mods**	**MMs size (avg)**	**MMs size (avg)**	**Size reduc. (%)**	**# Rules (avg)**	**Rules size (avg)**	**# Rules (avg)**	**Rules size (avg)**	**Size reduc. (%)**
Graphs	16	8	28 (3.5)	200 (12.5)	86.0%	10 (1.25)	63 (6.3)	56 (3.5)	760 (13.57)	91.7%
Networking	24	8	25 (3.13)	324 (13.5)	92.3%	15 (1.85)	58 (3.87)	108 (4.5)	936 (8.67)	93.8%
State machines	48	11	38 (3.45)	864 (18)	95.6%	13 (1.62)	68 (5.23)	160 (3.3)	2312 (14.45)	97.1%
Petri nets	64	13	49 (3.77)	1440 (22.5)	96.6%	27 (2.07)	125 (4.63)	768 (12)	5504 (7.16)	97.7%

Experiment setup. We created 4 behavioural LPLs: Networking (the running example); Graphs (with language variants like un-/directed edges and node hierarchy) along with a transformation implementing a breadth-first traversal; State machines (with variants like hierarchy, parallelism and time) plus a simulator; and Petri nets (with variants like bounded places, inhibitor and read arcs, and attribute or object tokens) plus a simulator. The examples are from the PL literature [9, 28], and are available at [6].

Results. Table 1 displays the results. The first two columns show the language family name and the number of configurations (i.e., language variants). The rest of the table shows the size reduction of the structural part (the meta-model) and the behavioural part (the rules) achieved by the LPLs, compared to an enumerative approach.

Regarding the structure (columns 3–6), the table reports: the number of modules in the LPL; the total (and average) meta-model size in each module, as given by the number of classes, attributes and references in each meta-model; the total (and average) meta-model size in each language variant of the enumerative approach; and the total meta-model size reduction (%) that our approach brings.

Regarding the behaviour (columns 7–11), the table shows: the number of Henshin rules in the LPL (and the average per module); the total (and average) size of the rules, as given by the number of objects, attributes and references in each rule; the number of rules in the enumerative approach (and the average per language variant); the total (and average) added size of all rules; and the total rule size reduction that the LPL approach brings.

Answering RQ1. In the study, our approach reduces the specification size of the structure by 86%–96.6%, and the rules size by 91.7%–97.7%. This reduction is correlated with the size of the language family (number of configurations/language variants).

Answering RQ2. In the study, adding a new module requires meta-models of size 3 to 4 (cf. column 3 of the table). This effort is considerably larger in the enumerative approach, where meta-model sizes range between 12.5 and 22.5 (column 5). Moreover, adding a new optional module implies doubling the number of language variants. For the semantics, each module has between 1.25 and 2.07 rules on average (column 7), with size between 3.87 and 6.3 (column 8). Instead, each variant in the enumerative approach requires between 3.3 and 12 rules (column 9), with size between 7.16 and 14.45 (column 10). Hence, extending a language family built with an enumerative approach requires creating more and bigger rules.

Threats to validity. The results are very promising, evincing substantial size reduction when defining a language family with our proposal. However, experiments with real developers are needed to assess the correlation between this size reduction and the actual effort to build the LPLs. We have used 4 LPLs. While we speculate that other language families may yield similar results (maintaining the correlation of size with number of configurations), more extensive experiments are needed for extra confidence in our claims.

8 RELATED WORK

We discuss our approach with respect to modelling language engineering, and review approaches to transforming rewrite rules.

Modelling language engineering. Product lines have been used to define the abstract syntax of language families concisely. For example, Merlin [16] supports definition of product lines of meta-models and the efficient analysis of their well-formedness properties. Merlin is not compositional, but it overlaps all meta-models in a so-called 150% meta-model, where elements attach formulae stating the configurations they belong to. MetaDepth [8] uses multi-level modelling to define language families, and product lines for their customization. None of these works considers semantics, and languages cannot be defined incrementally by composing modules.

Other proposals based on product lines consider semantics. Leduc *et al.* [27] define languages via extensible meta-models, and use the visitor pattern (combined with Java or an action language) for the semantics, hampering analysis. They also lack means, like our configurations, to select between possible language extensions. In [9], a 150% meta-model captures variability within a domain, and defines transformations on top in a modular way. Being based on a 150% meta-model, extension is challenging, and analysis is complex since transformations are written in EOL [24]. Méndez *et al.* [32] reverse engineer LPLs from DSL variants. Their LPLs consider syntax and operational semantics. Compatibility of operations is checked by comparing their signatures and ASTs, but truly behavioural analyses are difficult since operations are Java-like.

Regarding language composition, Durán and Zschaler [11, 12] combine definitions of languages and their rule-based behaviour to build more complex languages. While this is achieved using an amalgamation construction, akin to our Δ-rules, there is no notion of product line, language module, dependency, or configuration. Being based on graph transformation [50], their approach supports analysing whether the rule behaviour is protected at the level of traces, and not only on individual rules as we do. We will take inspiration on that approach for its application to product lines, where languages are composed in more intricate ways out of fragments.

Concern-oriented design [23] (an evolution of the reusable aspect models approach [22]) supports components encapsulating design concerns plus a feature model as the configuration interface. Concerns are composed incrementally via configuration selections, but compositional semantics is not considered. In the Kermeta language workbench [17], the operational semantics of languages is defined using the K3 meta-language in the form of aspects that are statically woven into the language syntax. While Kermeta enables

a modular language definition, it lacks an explicit variability model, which prevents the construction of language families. GEMOC Studio/Melange [10] improves the Kermeta workbench by supporting language families and variation points, but the analysis of the composed language is limited to type checking to ensure type safety.

Jurack *et al.* [18] define algebraic component concepts for (EMF) models with well-defined interfaces, but components lack a specification of their semantics. LanGems [47] uses roles to define interfaces for language modules, but lacks product-line capabilities. Moreover, semantics is specified by operations in metaclasses, which is challenging to analyse. MontiCore [4] and Neverlang [15] enable component-based language definition. However, they both deal with textual languages and use textual grammars for the abstract syntax, while we target meta-model-based languages.

Delta-oriented programming [41] permits building software products by defining a core module and a set of delta modules that specify changes to the core module. A software product is built by applying the delta modules to the core one. While highly expressive, ensuring confluence or semantic consistency is not possible in general. Hence, in the context of delta-oriented modelling, Pietsch et al. [38, 39] formalized delta operation as transformation rules and provided techniques for analysis, measurement and refactoring. Our approach works one meta-level up, but it would be interesting to consider their quality assurance techniques in our future work.

Rule transformation. Rule rewriting can be considered a particular realisation of rule inheritance/refinement, see [48] for a survey. Rule rewriting has also been studied in the literature. Parisi-Presicce [37] rewrites graph grammars by introducing high-level replacement systems where the productions are grammars and graph morphisms. Bottoni *et al.* [2] propose an incremental view on the syntax-directed construction of semantics of visual languages, which involves modifying triple graph rules where patterns declare conditions that a model must satisfy. The notion of higher-order transformation proposed in [29] permits applying transformations to rewriting rules, and obtain a valid graph rule. Our extension rules also rewrite rules, though their actions are limited to rule/NAC extension (i.e., they are non-deleting). Differently from the approaches mentioned above, variability-based rules [44] encode several rules into a single specification. While we could use variability-based rules to specify semantic variants, we opted for a modular approach to allow an incremental rule construction.

Transformations have not only been applied to rule rewriting, but also to product line rewriting. Taentzer *et al.* [45] use category theory to formalize the notion of transformation of software product lines, which combines modifications of feature models and product domain models. These transformations are proved to be sound. Instead, our extension rules modify rules, but preserve the domain meta-models and module dependencies.

9 CONCLUSIONS AND FUTURE WORK

We have presented a new modular approach to defining modelling language families, considering their abstract syntax and semantics. It is based on product line techniques, and involves the definition of language modules with interdependencies. Modules comprise a meta-model fragment, rules, and extension rules that expand the rules of other dependent modules. We have developed analysis for LPL behaviour consistency, demonstrated the applicability of the approach by an implementation atop Eclipse, and reported on an evaluation showing drastic size reductions.

In the future, we would like to apply our approach to industry cases and improve our tooling. On the theory side, we would like to lift existing analysis techniques for graph transformation (e.g., conflicts, dependencies [25]) to the product line level, develop finer granular analysis of (in-)consistency, devise effective testing techniques for LPLs, and consider richer meta-model notions (e.g., with OCL constraints, as in [16]). We will also consider generalizing the approach to support extension modules with several dependencies, whereby an LPL may span a directed acyclic graph instead of a tree [42], and enable modules with several alternative/OR groups.

APPENDIX: PROOF OF THEOREM 5.10

Model G in Fig. 8 is a pullback (PB) object, containing exactly the elements of G_ρ that are typed by the meta-model in module M_i. The spans $L \leftarrow K \rightarrow R$ and $L_{\rho(r)} \leftarrow K_{\rho(r)} \rightarrow R_{\rho(r)}$ of rules r and $\rho(r)$ are shown at the bottom, where morphisms l', k' and r' exist because $\rho(r)$ is an extension of r (cf. Def. 5.3). Spans $G \leftarrow D \rightarrow H$ and $G_\rho \leftarrow D_\rho \rightarrow H_\rho$ result from the direct derivations of r and $\rho(r)$. We need to show that: (1) morphism $m\colon L \rightarrow G$ exists and r is applicable, i.e., NACs are satisfied, (2) morphisms $d\colon D \rightarrow D_\rho$ and $h\colon H \rightarrow H_\rho$ exist, and (3) square (1) is PB.

(1) Morphism $m\colon L \rightarrow G$ exists since L is the PB object of $G \rightarrow G_\rho \leftarrow L_{\rho(r)}$. Indeed, on the one hand, $\rho(r)$ is a modular extension of r, so $L_{\rho(r)} \setminus L$ is typed by $MM_\rho \setminus MM(M_i)$. On the other hand, G only contains the elements of G_ρ typed by $MM(M_i)$. Hence, the PB object of $G \rightarrow G_\rho \leftarrow L_{\rho(r)}$ contains exactly the elements of $L_{\rho(r)}$ typed by $MM(M_i)$, i.e., L.
If $\rho(r)$ is applicable, all of its NACs are satisfied (G_ρ has no occurrence of them). NACs in $\rho(r)$ may either have been added by a NAC-rule, or have existed in r. In the first case, the NAC-rule should be a modular extension adding elements typed by $MM_\rho \setminus MM(M_i)$ and therefore not present in G. In the second case, the NAC of $\rho(r)$ may have been enlarged by modular extensions, whose elements cannot be in G either.

(2) Since $\rho(r)$ is a modular extension of r, both rules delete the same elements typed by $MM(M_i)$. In addition, $\rho(r)$ may delete more elements typed by $MM_\rho \setminus MM(M_i)$. Therefore, there must be a morphism $d\colon D \rightarrow D_\rho$.
Morphism $h\colon H \rightarrow H_\rho$ exists because of the universal pushout property. Since H is a pushout object, and we have $K \rightarrow D \rightarrow D_\rho \rightarrow H_\rho$ and $K \rightarrow R \rightarrow R_{\rho(r)} \rightarrow H_\rho$, there is a unique morphism $H \rightarrow H_\rho$ as required.

(3) Square (1) would not be a PB if the rule $\rho(r)$ would create elements typed by $MM_\rho \setminus MM(M_i)$. However, this is not possible since $\rho(r)$ is a modular extension.

ACKNOWLEDGMENTS

Work partially supported by Sapienza Visitor Programme, a visitor grant by Department of Computer Science at Sapienza, the Spanish Ministry of Science (PID2021-122270OB-I00) and the R&D programme of Madrid (P2018/TCS-4314). We thank the anonymous reviewers for their useful observations.

REFERENCES

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MoDELS (LNCS, Vol. 6394)*. Springer, 121–135.

[2] Paolo Bottoni, Esther Guerra, and Juan de Lara. 2008. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *Journal of Visual Languages & Computing* 19, 4 (2008), 429–455.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers.

[4] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A compositional framework for systematic modeling language reuse. In *MODELS*. ACM, 35–46.

[5] Capone-pl. 2022. https://capone-pl.github.io/.

[6] Capone-pl evaluation data. 2022. https://capone-pl.github.io/examples.html.

[7] Loris D'Antoni and Margus Veanes. 2021. Automata modulo theories. *Commun. ACM* 64, 5 (2021), 86–95.

[8] Juan de Lara and Esther Guerra. 2021. Language family engineering with product lines of multi-level models. *Formal Aspects of Computing* 33, 6 (2021), 1173–1208.

[9] Juan de Lara, Esther Guerra, Marsha Chechik, and Rick Salay. 2018. Model transformation product lines. In *MoDELS*. ACM, 67–77.

[10] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A meta-language for modular and reusable development of DSLs. In *SLE*. ACM, 25–36.

[11] Francisco Durán, Antonio Moreno-Delgado, Fernando Orejas, and Steffen Zschaler. 2017. Amalgamation of domain specific languages with behaviour. *J. Log. Algebraic Methods Program.* 86, 1 (2017), 208–235.

[12] Francisco Durán, Steffen Zschaler, and Javier Troya. 2012. On the Reusable Specification of Non-functional Properties in DSLs. In *Software Language Engineering, 5th International Conference, SLE (Lecture Notes in Computer Science, Vol. 7745)*. Springer, 332–351.

[13] Eclipse. 2022. https://www.eclipse.org/.

[14] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.

[15] Luca Favalli, Thomas Kühn, and Walter Cazzola. 2020. Neverlang and FeatureIDE just married: Integrated language product line development environment. In *SPLC*. ACM, 1–11.

[16] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2022. Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Softw. Eng.* 48, 2 (2022), 397–416.

[17] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Softw. Syst. Model.* 14, 2 (2015), 905–920.

[18] Stefan Jurack and Gabriele Taentzer. 2010. A component concept for typed graphs with inheritance and containment structures. In *ICGT (LNCS, Vol. 6372)*. Springer, 187–202.

[19] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report CMU/SEI-90-TR-021. Carnegie Mellon Universit.

[20] Nadine Kashmar, Mehdi Adda, and Mirna Atieh. 2020. From access control models to access control metamodels: A survey. In *FICC (LNNS, Vol. 70)*. Springer, 892–911.

[21] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley.

[22] Jörg Kienzle, Wisam Al Abed, Franck Fleurey, Jean-Marc Jézéquel, and Jacques Klein. 2010. *Aspect-Oriented Design with Reusable Aspect Models.* Springer Berlin Heidelberg, Berlin, Heidelberg, 272–320.

[23] Jörg Kienzle, Gunter Mussbacher, Philippe Collet, and Omar Alam. 2016. Delaying decisions in variable concern hierarchies. In *GPCE*. ACM, 93–103.

[24] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *ECMDA-FA (LNCS, Vol. 4066)*. Springer, 128–142.

[25] Leen Lambers, Daniel Strüber, Gabriele Taentzer, Kristopher Born, and Jevgenij Huebert. 2018. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *Proc. ICSE*. ACM, 716–727.

[26] Saunders Mac Lane. 1971. *Categories for the Working Mathematician.* Springer.

[27] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. 2017. Revisiting visitors for modular extension of executable DSMLs. In *MODELS*. IEEE Computer Society, 112–122.

[28] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GCBSE*, Jan Bosch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 10–24.

[29] Rodrigo Machado, Leila Ribeiro, and Reiko Heckel. 2015. Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars. *Theor. Comp. Sci.* 594 (2015), 1–23.

[30] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What industry needs from architectural languages: A survey. *IEEE Trans. Soft. Eng.* 39, 6 (2013), 869–891.

[31] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer.

[32] David Méndez-Acuña, José Angel Galindo, Benoît Combemale, Arnaud Blouin, and Benoit Baudry. 2017. Reverse engineering language product lines from existing DSL variants. *J. Syst. Softw.* 133 (2017), 145–158.

[33] David Méndez-Acuña, José Angel Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. *Comput. Lang. Syst. Struct.* 46 (2016), 206–235.

[34] T. Murata. 1989. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[35] L. Northrop and P. Clements. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[36] OSGi working group. 2022. OSGi: The dynamic module system for Java. https://www.osgi.org/.

[37] Francesco Parisi-Presicce. 1994. Transformations of graph grammars. In *TAGT (LNCS, Vol. 1073)*. Springer, 428–442.

[38] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal foundations for analyzing and refactoring delta-oriented model-based software product lines. In *Proc. of the 23rd International Systems and Software Product Line Conference, SPLC*. ACM, 30:1–30:11.

[39] Christopher Pietsch, Dennis Reuling, Udo Kelter, and Timo Kehrer. 2017. A tool environment for quality assurance of delta-oriented model-based SPLs. In *Proc. 11tt International Workshop on Variability Modelling of Software-intensive Systems, VaMoS*, Maurice H. ter Beek, Norbert Siegmund, and Ina Schaefer (Eds.). ACM, 84–91.

[40] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, Berlin, Heidelberg.

[41] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *SPLC (LNCS, Vol. 6287)*. Springer, 77–91.

[42] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Comput. Networks* 51, 2 (2007), 456–479.

[43] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional.

[44] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: Formal foundation and application. *Formal Asp. Comput.* 30, 1 (2018), 133–162.

[45] Gabriele Taentzer, Rick Salay, Daniel Strüber, and Marsha Chechik. 2017. Transformations of software product lines: A generalizing framework based on category theory. In *MoDELS*. IEEE, 101–111.

[46] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. 2009. On the use of higher-order model transformations. In *ECMDA-FA (LNCS, Vol. 5562)*. Springer, 18–33.

[47] Christian Wende, Nils Thieme, and Steffen Zschaler. 2009. A Role-Based Approach towards Modular Language Engineering. In *Software Language Engineering, Second International Conference, SLE (LNCS, Vol. 5969)*. Springer, 254–273.

[48] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. 2012. Surveying Rule Inheritance in Model-to-Model Transformation Languages. *J. Object Technol.* 11, 2 (2012), 3: 1–46.

[49] Xtext. 2022. https://www.eclipse.org/Xtext/.

[50] Steffen Zschaler and Francisco Durán. 2021. *GTSMorpher: Safely Composing Behavioural Analyses Using Structured Operational Semantics.* Springer International Publishing, Cham, 189–215.